CHAPTER 2

The System-on-Chip Design Process

This chapter gives an overview of the System-on-Chip (SoC) design methodology. The topics include:

- Canonical SoC design
- System design flow
- The role of specifications throughout the life of a project
- Steps for system design process

2.1 A Canonical SoC Design

Consider the chip design in Figure 2-1 on page 10. We claim that, in some sense, this design represents a canonical or generic form of an SoC design. It consists of:

- A microprocessor and its memory subsystem
- On-chip buses (high-speed and low-speed) to provide the datapath between cores
- A memory controller for external memory
- A communications controller
- A video decoder
- A timer and interrupt controller
- A general purpose I/O (GPIO) interface
- A UART interface

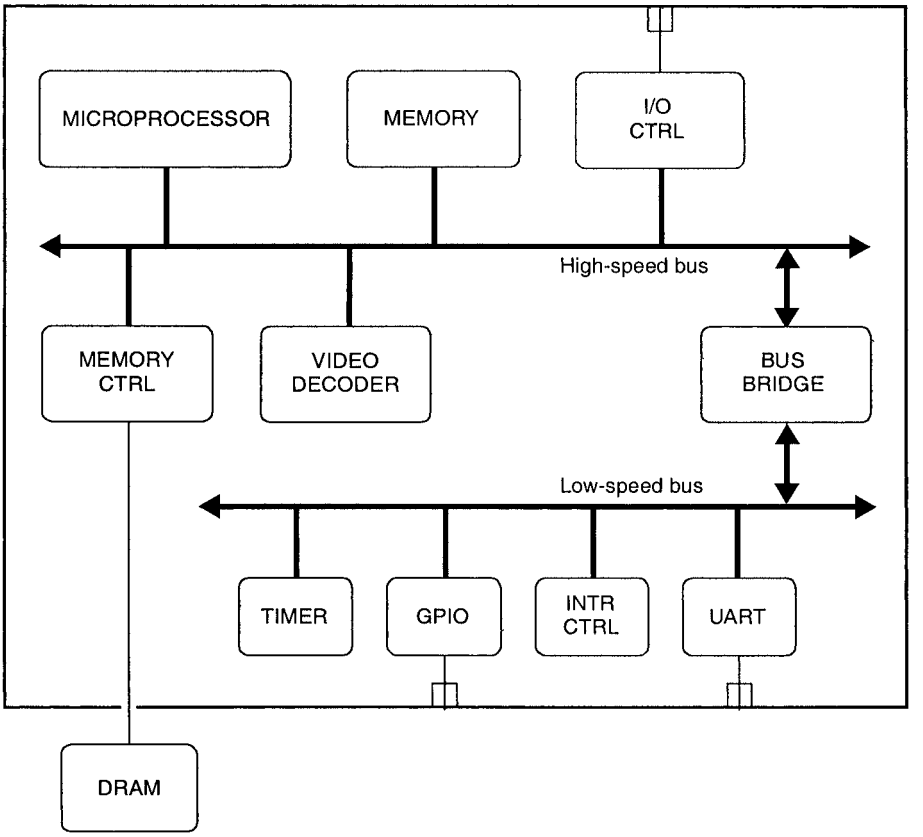

Figure 2-1 Canonical hardware view of SoC

This design is somewhat artificial, but it contains most of the structures and challenges found in real SoC designs. For example:

- The microprocessor could be anything from an 8-bit 8051 to a 64-bit RISC.
- The memory subsystem could be single- or multi-level, and could include SRAM and/or DRAM.
- The external memory could be DRAM (shown), SRAM, or Flash.
- The I/O controller could include PCI, PCI-X, Ethernet, USB, IEEE 1394, analog-to-digital, digital-to-analog, electro-mechanical, or electro-optical converters.
- The video decoder could be MPEG, ASF, or AVI.
- The GPIO could be used for powering LEDs or sampling data lines.

The design process required to specify such a system—to develop and verify the blocks, and assemble them into a fabricated chip—contains all the basic elements and challenges of an SoC design.

Real SoC designs are, of course, much more complex than this canonical example. A real design would typically include several sets of IP interfaces and data transformations. Many SoC designs today include multiple processors, and combinations of processors and DSPs. The memory structures of SoC designs are often very complex as well, with various levels of caching and shared memory, and specific data structures to support data transformation blocks, such as the video decoder. Thus, the canonical design is just a miniature version of an SoC design that allows us to discuss the challenges of developing these chips utilizing reusable macros.

2.2 System Design Flow

To meet the challenges of SoC, chip designers are changing their design flows in two major ways:

- From a waterfall model to a spiral model
- From a top-down methodology to a combination of top-down and bottom-up

2.2.1 Waterfall vs. Spiral

The traditional model for ASIC development, shown in Figure 2-2 on page 12, is often called a *waterfall model*. In a waterfall model, the project transitions from phase to phase in a step function, never returning to the activities of the previous phase. In this model, the design is often tossed "over the wall" from one team to the next without much interaction between the teams.

This process starts with the development of a specification for the ASIC. For complex ASICs with high algorithmic content, such as graphics chips, the algorithm may be developed by a graphics expert; this algorithm is then given to a design team to develop the RTL for the SoC.

After functional verification, either the design team or a separate team of synthesis experts synthesizes the ASIC into a gate-level netlist. Then timing verification is performed to verify that the ASIC meets timing. Once the design meets its timing goals, the netlist is given to the physical design team, which places and routes the design. Finally, a prototype chip is built and tested. This prototype is delivered to the software team for software debug.

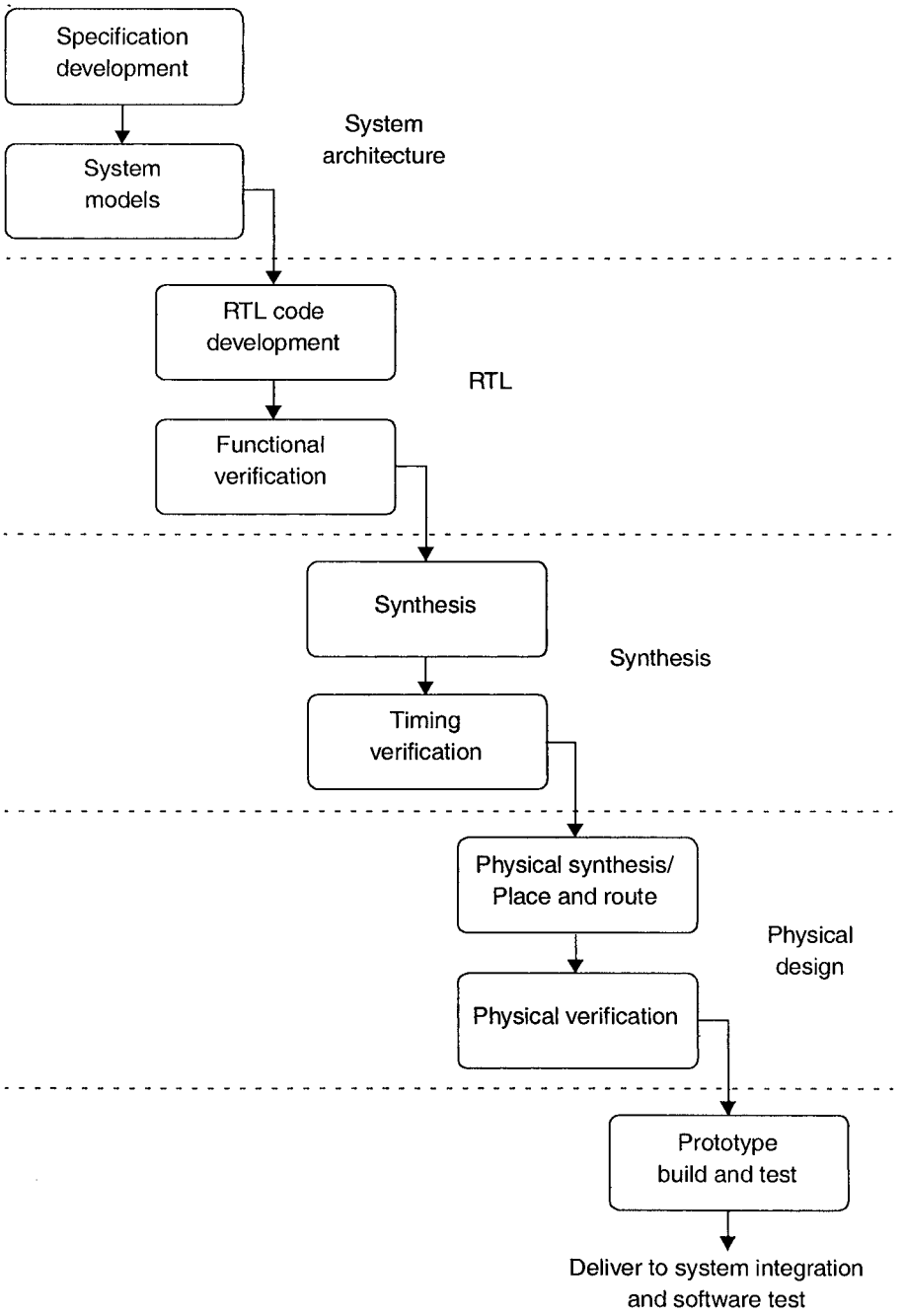

Figure 2-2 Traditional waterfall ASIC design flow

In most projects, software development is started shortly after the hardware design is started. But without a model of the hardware to use for debug, the software team can make little real progress until the prototype is delivered. Thus, hardware and software development are essentially serialized.

This flow has worked well in designs of up to 100K gates and down to 0.5 μm. It has consistently produced chips that worked right the first time, although often the systems that were populated with them did not. But this flow has always had problems because handoffs from one team to the next are rarely clean. For example, the RTL design team may have to go back to the system designer and tell him that the algorithm is not implementable, or the synthesis team may have to go back to the RTL team and inform them that the RTL must be modified to meet timing.

For large, deep submicron designs, this waterfall methodology simply does not work. Large systems have sufficient software content that hardware and software must be developed concurrently to ensure correct system functionality. Physical design issues must be considered early in the design process to ensure that the design can meet its performance goals.

As complexity increases, geometry shrinks, and time-to-market pressures continue to escalate, chip designers are turning to a modified flow to produce today's larger SoC designs. Many teams are moving from the old waterfall model to the newer *spiral development model.* In the spiral model, the design team works on multiple aspects of the design simultaneously, incrementally improving in each area as the design converges on completion.

Figure 2-3 on page 14 shows the spiral SoC design flow. This flow is characterized by:

- Parallel, concurrent development of hardware and software
- Parallel verification and synthesis of modules
- Floorplanning and place-and-route included in the synthesis process
- Modules developed only if a pre-designed hard or soft macro is not available
- Planned iteration throughout

In the most aggressive projects, engineers simultaneously develop top-level system specifications, algorithms for critical subblocks, system-level verification suites, and timing budgets for the final chip integrations. That means that they are addressing all aspects of hardware and software design concurrently: functionality, timing, physical design, and verification.

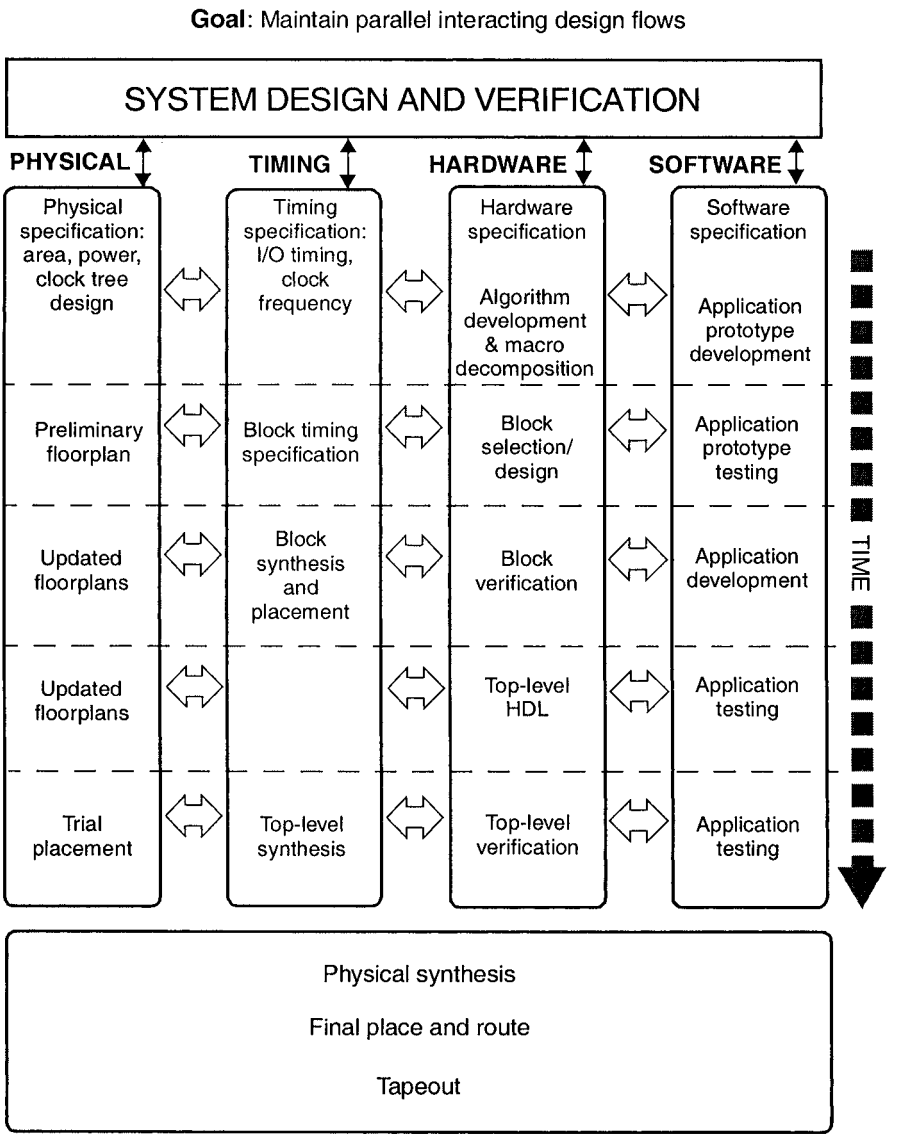

Figure 2-3 Spiral SoC design flow

2.2.2 Top-Down vs. Bottom-Up

The classic top-down design process can be viewed as a recursive routine that begins with specification and decomposition, and ends with integration and verification:

1. Write complete specifications for the system or subsystem being designed.
2. Refine its architecture and algorithms, including software design and hardware/software cosimulation if necessary.
3. Decompose the architecture into well-defined macros.
4. Design or select macros; this is where the recursion occurs.
5. Integrate macros into the top level; verify functionality and timing.
6. Deliver the subsystem/system to the next higher level of integration; at the top level, this is tapeout.
7. Verify all aspects of the design (functionality, timing, etc.).

With increasing time-to-market pressures, design teams have been looking at ways to accelerate this process. Increasingly powerful tools, such as synthesis and emulation tools, have made significant contributions. Developing libraries of reusable macros also aids in accelerating the design process.

But, like the waterfall model of system development, the top-down design methodology is an idealization of what can really be achieved. A top-down methodology assumes that the lowest level blocks specified can, in fact, be designed and built. If it turns out that a block is not feasible to design, the whole specification process has to be repeated. For this reason, real world design teams usually use a mixture of top-down and bottom-up methodologies, building critical low-level blocks while they refine the system and block specifications. Libraries of reusable hard and soft macros clearly facilitate this process by providing a source of pre-verified blocks, proving that at least some parts of the design can be designed and fabricated in the target technology and perform to specification.

2.2.3 Construct by Correction

The Sun Microsystems engineers that developed the UltraSPARC processor have described their design process as "construct by correction." In this project, a single team took the design from architectural definition through place and route. In this case, the engineers had to learn how to use the place and route tools, whereas, in the past, they had always relied on a separate team for physical design. By going through the entire flow, the team was able to see for themselves the impact that their architectural decisions had on the area, power, and performance of the final design.

The UltraSPARC team made the first pass through the design cycle—from architecture to layout—as fast as possible, allowing for multiple iterations through the entire process. By designing an organization and a development plan that allowed a single

group of engineers to take the design through multiple complete iterations, the team was able to see their mistakes, correct them, and refine the design several times before the chip was finally released to fabrication. The team called this process of iteration and refinement "construct by correction."

This process is the opposite of "correct by construction" where the intent is to get the design completely right during the first pass. The UltraSPARC engineers believed that it was not possible at the architectural phase of the design to foresee all the implication their decisions would have on the final physical design.

The UltraSPARC development project was one of the most successful in Sun Microsystems' history. The team attributes much of its success to the "construct by correction" development methodology.

2.2.4 Summary

Hardware and software teams have consistently found that iteration is an inevitable part of the design process. There is significant value in planning for iteration, and developing a methodology that minimizes the overall design time. This usually means minimizing the number of iterations, especially in major loops. Going back to the specification after an initial layout of a chip is expensive; we want to do it as few times as possible, and as early in the design cycle as possible.

We would prefer to iterate in tight, local loops, such as coding, verifying, and synthesizing small blocks. These loops can be very fast and productive. We can achieve this if we can plan and specify the blocks we need with confidence that they can be built to meet the needs of the overall design. A rich library of pre-designed blocks clearly helps here; parameterized blocks that allow us to make tradeoffs between function, area, and performance are particularly helpful.

In the following sections we describe design processes in flow diagrams because they are a convenient way of representing the process steps. Iterative loops are often not shown explicitly to simplify the diagrams. But we do not wish to imply a waterfall methodology—that one stage cannot be started until the previous one is finished. Often, it is necessary to investigate some implementation details before completing the specification. Rather, it is our intention that no stage can be considered complete until the previous stage is completed.

A word of caution: the inevitability of iteration should never be used as an excuse to short-change the specification process. Spending time in carefully specifying a design is the best way to minimize the number of iterative loops and to minimize the amount of time spent in each loop.

2.3 The Specification Problem

The first part of the design process consists of recursively developing, verifying, and refining a set of specifications until they are detailed enough to allow RTL coding to begin. The rapid development of clear, complete, and consistent specifications is a difficult problem. In a successful design methodology, it is the most crucial, challenging, and lengthy phase of the project. If you know what you want to build, implementation mistakes are quickly spotted and fixed. If you don't know, you may not spot major errors until late in the design cycle or until fabrication.

Similarly, the cost of documenting a specification during the early phases of a design is much less than the cost of documenting it after the design is completed. The extra discipline of formalizing interface definitions, for instance, can occasionally reveal inconsistencies or errors in the interfaces. On the other hand, documenting the design after it is completed adds no real value for the designer and either delays the project or is skipped altogether.

2.3.1 Specification Requirements

Specifications describe the behavior of a system; more specifically, they describe how to manipulate the interfaces of a system to produce the desired behavior. In this sense, specifications are largely descriptions of interfaces. Functional specifications describe the interfaces of a system or block as seen by the user of the systems. They describe the pins, buses, and registers, and how these can be used to manipulate data. Architectural specifications describe the interfaces between component parts and how transactions on these interfaces coordinate the functions of each block, and create the desired system-level behavior.

In an SoC design, specifications are required for both the hardware and software portions of the design. The specifications must completely describe all the interfaces between the design and its environment, including:

Hardware

- Functionality
- External interfaces to other hardware (pins, buses, and how to use them)
- Interface to SW (register definitions)
- Timing
- Performance
- Physical design issues such as area and power

Software

- Functionality
- Timing
- Performance
- Interface to HW
- SW structure, kernel

Traditionally, specifications have been written in a natural language, such as English, and have been plagued by ambiguities, incompleteness, and errors. Many companies, realizing the problems this causes, have started using executable specifications for some or all of the system.

2.3.2 Types of Specifications

There are two major techniques currently being used to help make hardware and software specifications more robust and useful: *formal specification* and *executable specification*.

- **Formal specifications –** In formal specifications, the desired characteristics of the design are defined independently of any implementation. Once a formal specification is generated for a design, formal methods such as property checking can be used to prove that a specific implementation meets the requirements of the specification. A number of formal specification languages have been developed, including one for VHDL called VSPEC [1]. These languages typically provide a mechanism for describing not only functional behavior, but timing, power, and area requirements as well. To date, formal specification has not been used widely for commercial designs, but continues to be an important research topic and is considered promising in the long term.

- **Executable specifications –** Executable specifications are currently more useful for describing functional behavior in most design situations. An executable specification is typically an abstract model for the hardware and/or software being specified. For high-level specifications, it is typically written in C, C++, or some variant of C++, such as SystemC or a Hardware Verification Language (HVL). At the lower levels, hardware is usually described in Verilog or VHDL. Developing these models early in the design process allows the design team to verify the basic functionality and interfaces of the hardware and software long before the detailed design begins.

  Most executable specifications address only the functional behavior of a system, so it may still be necessary to describe critical physical specifications—timing, clock frequency, area, and power requirements—in a written document. Efforts are under way to develop more robust forms of capturing timing and physical design requirements.

2.4 The System Design Process

Many chip designs are upgrades or modifications of an existing design. For these chips, the architecture can be reasonably obvious. But for SoC designs with significant new content, system design can be a daunting process. Determining the optimal architecture in terms of cost and performance involves a large number of complex decisions and tradeoffs, such as:

- What goes in software and what goes in hardware
- What processor(s) to use, and how many
- What bus architecture is required to achieve the required system performance
- What memory architecture to use to reach an appropriate balance between power, area, and speed

In most SoC designs, it is not possible, purely by analysis, to develop a system architecture that meets the design team's cost and performance objectives. Extensive modeling of several alternative architectures is often required to determine an appropriate architecture. The system design process consists of proposing a candidate system design and then developing a series of models to evaluate and refine the design.

The system design process shown in Figure 2-4 on page 22 employs both executable and written specifications to specify and refine a system architecture. This process involves the following steps:

1. **Create the system specification**

   The process begins by identifying the objectives of the design; that is, the *system requirements*: the required functions, performance, cost, and development time for the system. These are formulated into a *preliminary specification*, often written jointly by engineering and marketing.

2. **Develop a behavioral model**

   The next step is to develop an initial high-level design and create a *high-level behavioral model* for the overall system. This model can be used to test the basic algorithms of the system design and to show that they meet the requirements outlined in the specification. For instance, in a wireless communication design it may be necessary to demonstrate that the design can meet certain performance levels in a noisy environment. Or a video processing design may need to demonstrate that losses in compression/decompression are at an acceptable level.

   This high-level model provides an executable specification for the key functions of the system. It can then be used as the reference for future versions of the design. For instance, the high-level model and the detailed design of a video chip can be given the same input stream, and the output frames can be compared to verify that the detailed design products the expected result.

3. Refine and test the behavioral model

A verification environment for the high-level model is developed to *refine and test* the algorithm. This environment provides a mechanism for refining the high-level design, verifying the functionality and performance of the algorithm. If properly designed, it can also be used later to verify models for the hardware and software, such as an RTL model verified using hardware/software cosimulation. For systems with very high algorithmic content, considerable model development, testing, and refinement occurs before the hardware/software partitioning.

For instance, a graphics or multimedia system may be initially coded in C/C++ with all floating-point operations. This approach allows the system architect to code and debug the basic algorithm quickly. Once the algorithm is determined, a fixed-point version of the model is developed. This allows the architect to determine what accuracy is required in each operation to achieve performance goals while minimizing die area.

Finally, a cycle-accurate and bit-accurate model is developed, providing a very realistic model for implementation. In many system designs, this refinement of the model from floating-point to fixed-point to cycle-accurate is one of the key design challenges.

These multiple models are very useful when the team is using hardware/software cosimulation to debug their software. The behavioral model can provide simulation for most development and debugging. Later, the detailed, cycle-accurate model can be used for final software debug.

4. Determine the hardware/software partition (decomposition)

As the high-level model is refined, the system architects determine the *hardware/software partition*; that is, the division of system functionality between hardware and software. This is largely a manual process requiring judgment and experience on the part of the system architects and a good understanding of the cost/performance tradeoffs for various architectures. A rich library of pre-verified, characterized macros and a rich library of reusable software modules are essential for identifying the size and performance of various hardware and software functions.

The final step in hardware/software partitioning is to define the interfaces between hardware and software, and specify the communication protocols between them.

5. Specify and develop a hardware architectural model

Once the requirements for the hardware are defined, it is necessary to specify a detailed hardware architecture. This involves determining which hardware blocks will be used and how they will communicate. Memory architecture, bus structure and bus bandwidth can be critical issues. Most SoC chips have many different blocks communicating over one or more buses. The traffic over these buses and

thus the required bandwidth can be very application dependent, so it becomes necessary to evaluate architectures by running substantial amounts of representative application code on them.

Running significant amounts of application code on an RTL-level design is often too time consuming to be practical. To address this problem, designers are using transaction-level models to model interfaces and bus behavior. By eliminating the detailed behavior of pins and signals on a bus or interface, these models can run considerably faster than RTL models, yet still give accurate estimates of performance. Many of the features of the SystemC language were developed explicitly to facilitate transaction-level modeling.

Determining the final hardware architecture consists of developing, testing, and modifying architectural-level models of the system until an architecture is demonstrated to meet the system requirements.

6. Refine and test the architectural model (cosimulation)

One of the classic problems in system design is that software development often starts only once the hardware has been built. This serialization of hardware and software development has led to many delayed or even cancelled projects.

The architectural model for the system can be used for hardware/software cosimulation. It provides sufficient accuracy that software can be developed and debugged on it, long in advance of getting actual hardware.

As the software content of systems continues to grow, hardware/software co-development and cosimulation will become increasingly critical to the success of SoC projects. Having fast, accurate models of the hardware will be key to this aspect of SoC design.

7. Specify implementation blocks

The output of the architectural exploration activity is a *hardware specification*: a detailed specification of the functionality, performance, and interfaces for the hardware system and its component blocks.

In its written form, the hardware specification includes a description of the basic functions, the timing, area, and power requirements, and the physical and software interfaces, with detailed descriptions of the I/O pins and the register map.

The architectural model itself functions as an executable specification for the hardware.

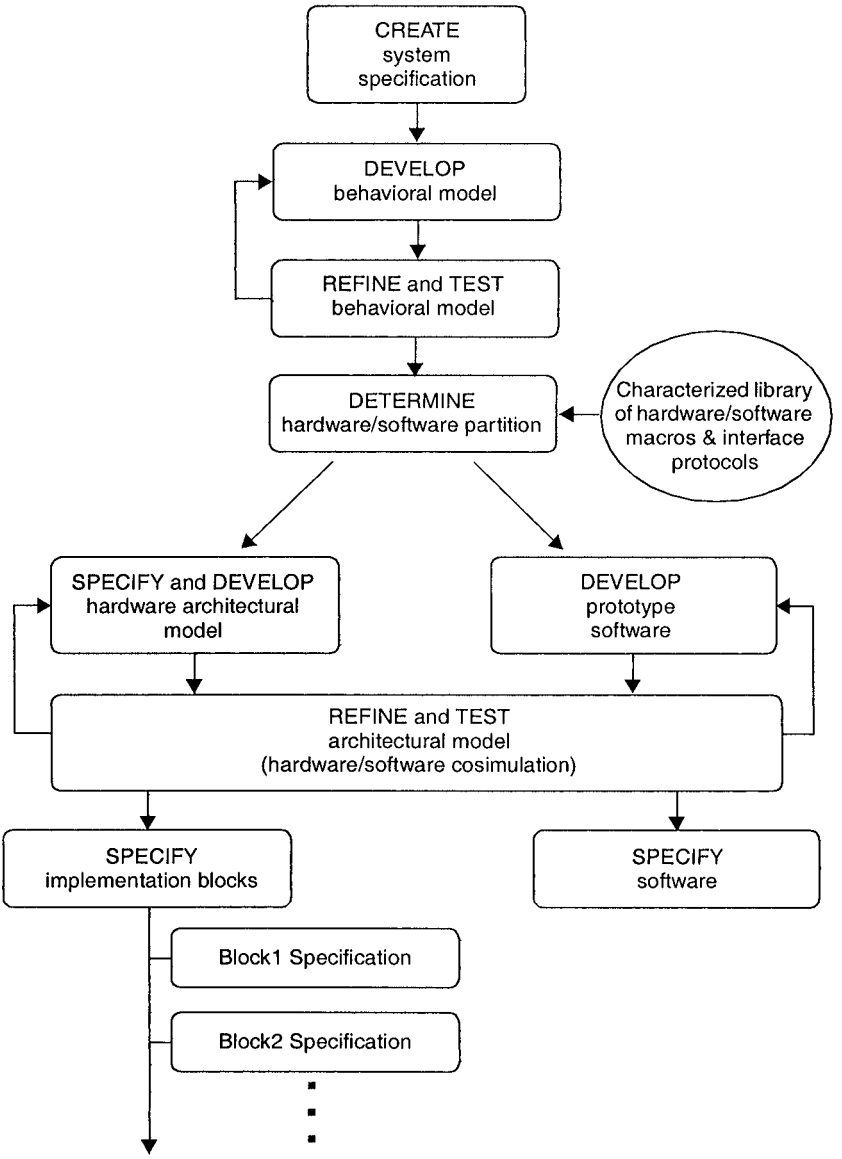

Figure 2-4 Top-level system design and recommended applications for each step

References

1. Ellsberger, Jan et al. *Sdl: Formal Object-Oriented Language for Communicating Systems.* Prentice Hall, 1997.

http://www.springer.com/978-0-387-74098-0

Reuse Methodology Manual for System-on-a-Chip Designs

Bricaud, P.

2002, XX, 292 p., Softcover
ISBN: 978-0-387-74098-0